Fra problem til program

Gitt et problem, hvordan går man fram for å programmere en løsning?

- UML klassediagrammer
- Enhetstesting
- Dokumentasjon

Som student ønsker vi oss et program som kan holde oversikt over de eksamene vi har tatt. Det skal kunne skrive ut ut en liste over alle eksamenene eller bare de som er tatt i et gitt år og regne ut gjennomsnittskarakteren. Data (dvs all informasjonen om eksamene) ligger i en fil.

Filformatet

Hver eksamen bruker 3–5 linjer:

1. Emnet
2. Karakter
3. Dato (enten «-» for i dag eller tre linjer med dag, måned og år)

```
INF1000
B
12
06
2011
INF1010
D
06
11
2012
INF2220
A
-
```

Hvilke klasser trenger jeg?

Når jeg skal planlegge hvilke klasser jeg trenger og hvordan de henger sammen, kan en tegning hjelpe.

Ved samarbeid er det lurt å tegne på samme måte.

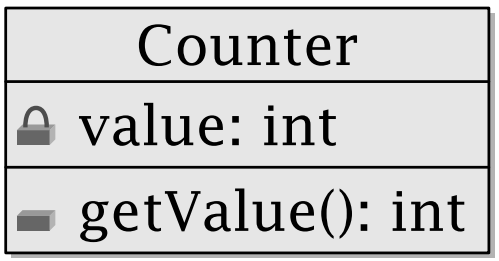
Unified Modeling Language

UML er den meste brukte standarden for å illustrere systemutvikling. Av ca 20 typer diagrammer skal vi bruke **klassediagrammer** med angivelse av forhold («association»).

- Alle skjønner notasjonen.
- Det finnes verktøy for enkelt å lage diagrammer.
- Utifra UML-diagrammer kan man automatisk lage mye av programkoden.

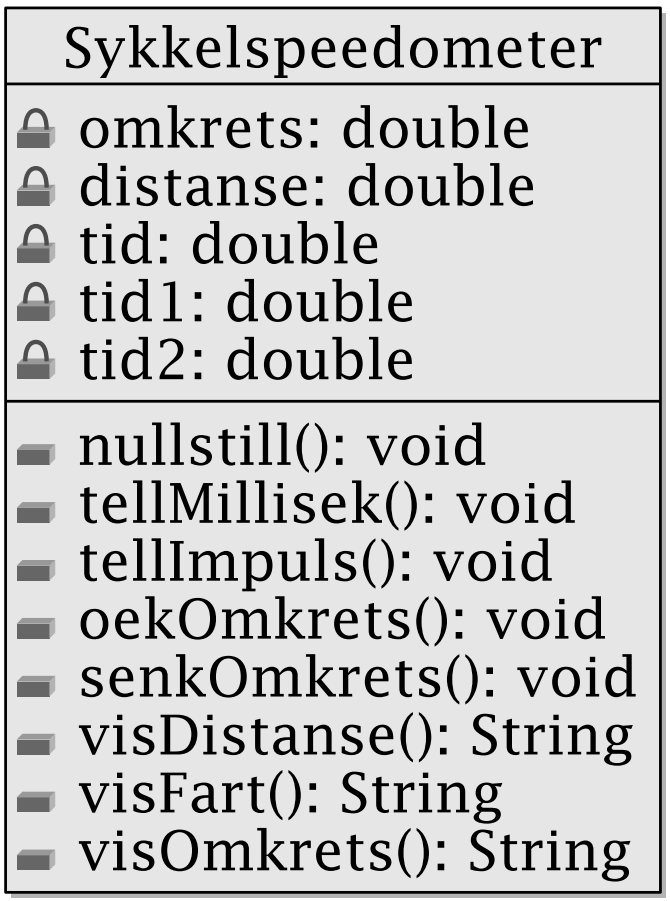
Enkeltklasser

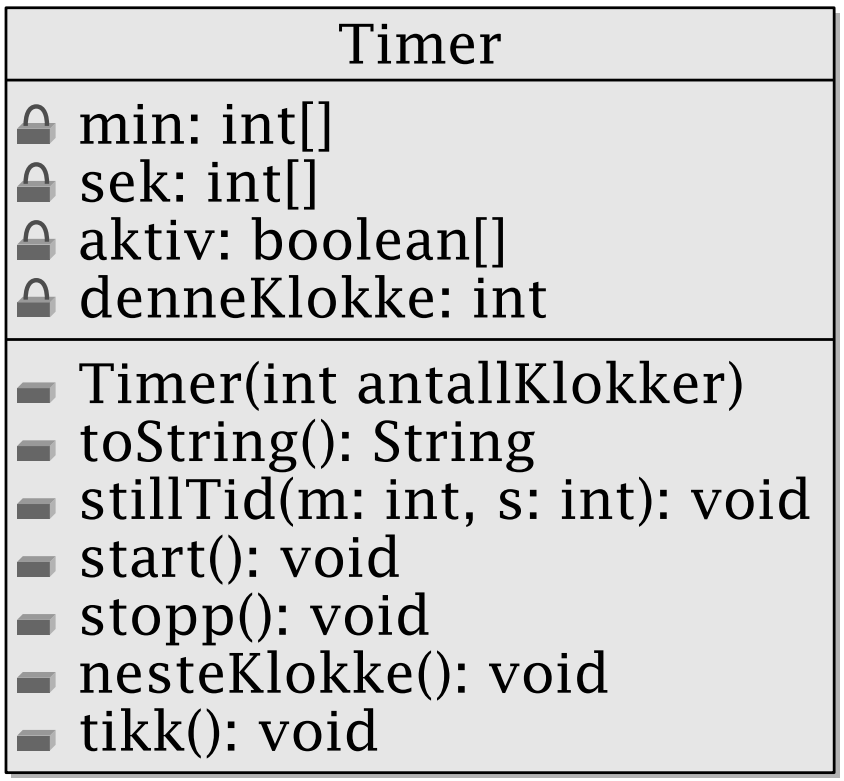
En klasse tegnes som en firkant med tre deler adskilt med en strek:



1. Klassens navn
2. Representasjonen (dvs objektvariablene); en lås (eller «-») viser at de er **private**.
3. Grensesnittet (dvs objektmetodene); en helt åpen lås (eller «+») viser at de kan brukes utenfra (dvs er **public**).

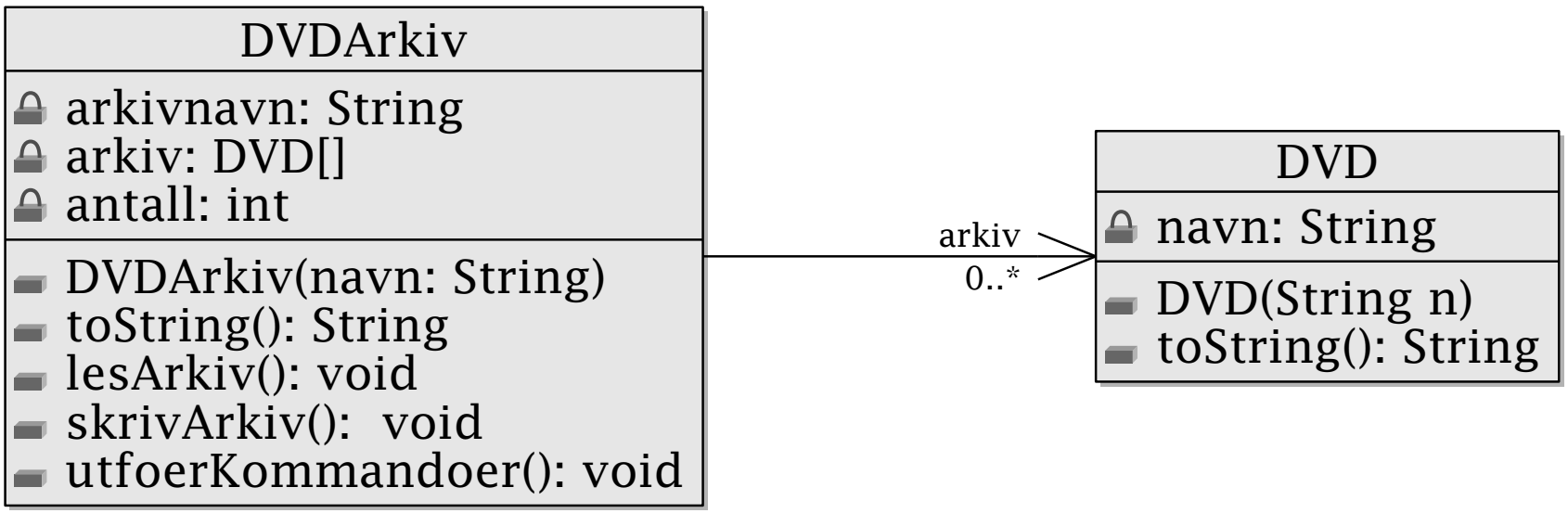
Noen eksempler





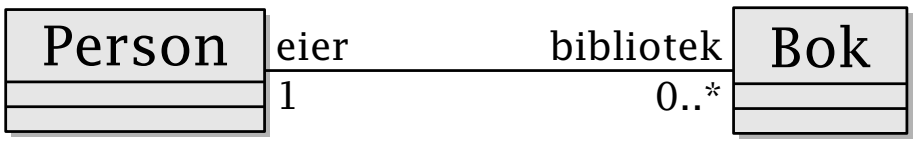
Klasser i sammenheng

Det er vel så viktig å angi hva som er *sammenhengen* mellom klassene.



En pil forteller at et DVDArkiv refererer til DVD-er. Navnet angir hva referansen er. Tallet angir antallet.

Hvis det er referanser begge veier, bruker vi en strek:



leses slik:

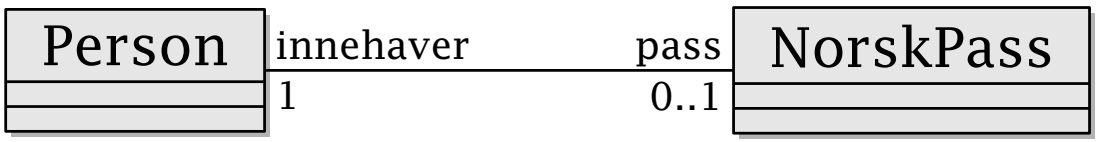
1. En Person har et *bibliotek* med 0 eller flere Bok-er.
2. En Bok har en *eier* som er nøyaktig 1 Person.

NB!

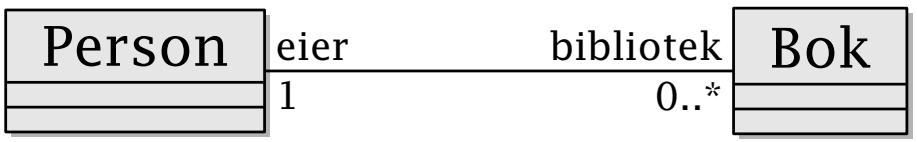
Vi tar alltid utgangspunkt i *ett* objekt!

Det er i praksis tre slags antall i bruk:

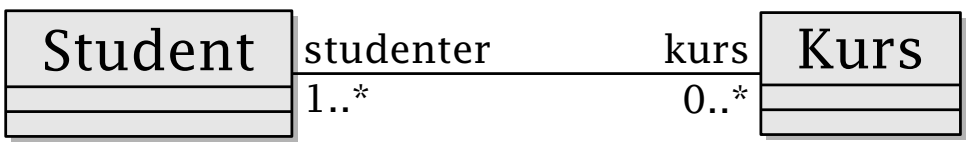
1-til-1



1-til-mange



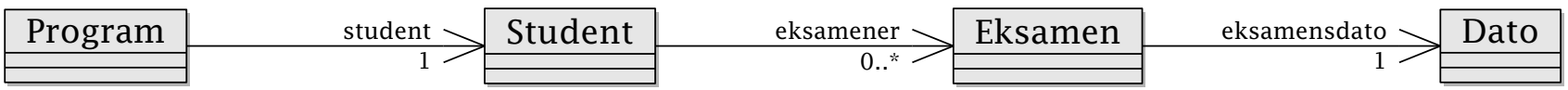
mange-til-mange



Fase 1: Hvilke klasser trenger vi?

Klasser er mønstre for objekter (dvs ting eller konsepter) vi skal jobbe med.

Et godt forslag: Hvilke substantiver er brukt i oppgaveteksten?



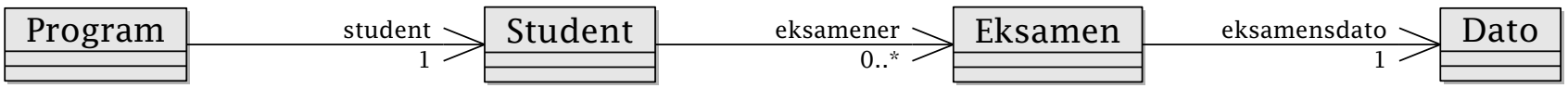
Noen oppklaringer

- Må jeg forenkle virkeligheten?
  Ja. Alltid.
- Finnes det én riktig fasit?
  Nei; mange løsninger er like gode (og mange er ganske dårlige!).
- Hvordan vet jeg om forslaget mitt er bra?
  Hvis resten av programmeringen går greit, er det en god løsning.
- Hva gjør jeg hvis det viser seg at forslaget mitt er dårlig?
  Gå tilbake og endre det forrige forslaget.

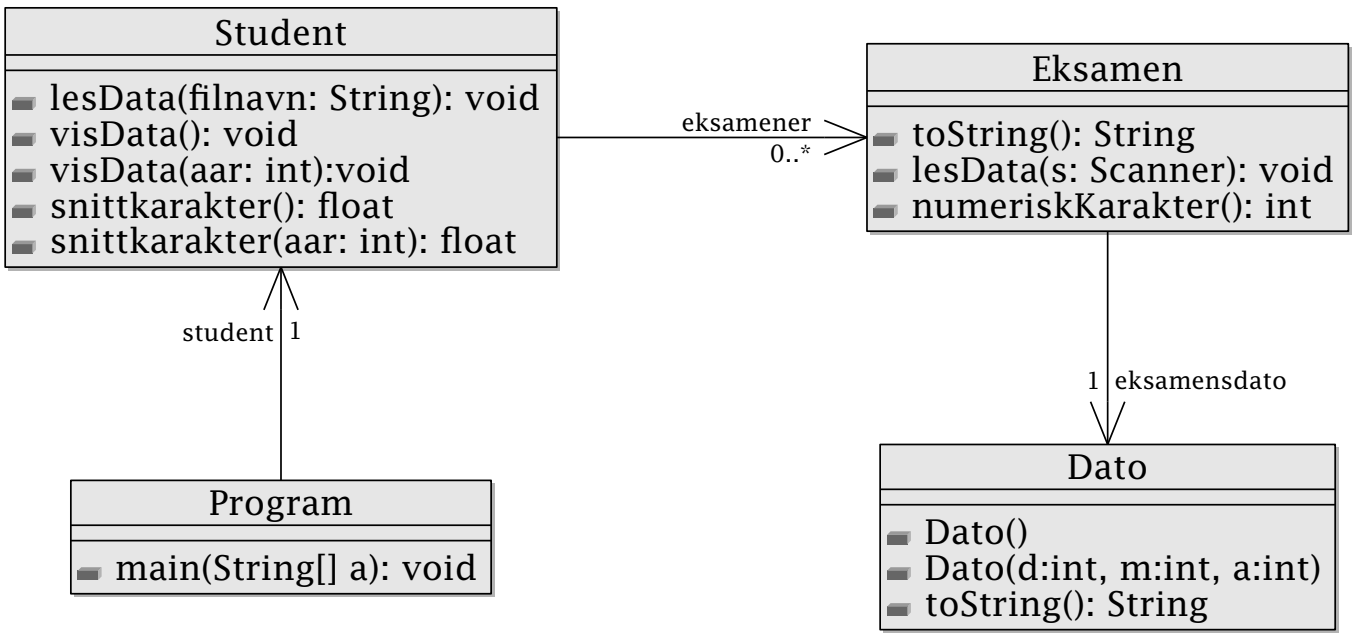
Resten av implementasjonen

2. Angi grensesnittet
3. Finn representasjonen
4. Skriv koden
5. Test koden
6. Skriv dokumentasjon

Fase 2: Grensesnittet



Med grensnitt blir tegningen slik:



Enhetstesting

For å være rimelig trygg på at en klasse er korrekt programmert, bør vi teste den *alene*.

Skriv et lite hovedprogram (dvs en klasse med main) som oppretter objekter og tester dem.

Husk!

Du har programmert klassen, så du er den nærmeste til å avsløre feil og mangler.

Dokumentasjon

En meget viktig del av programmeringen er dokumentasjon.

- Andre kan bruke programkoden vår.
- Andre må kanskje jobbe videre med koden vår, eller finne feil.
- Vi trenger den selv når vi har glemt hva vi gjorde og hvorfor.
- Å skrive dokumentasjon påvirker programmene vi skriver. («Er det vanskelig å skrive kommentarer, er løsningen uklart tenkt.»)

Hvordan skrive JavaDoc-kommentarer?

Kommentarene (i HTML-kode!) for klasser ser slik ut:

```
/**
 * Én setning som kort beskriver klassen
 * Mer forklaring
 *      ⋮
 * @author  navn
 * @author  navn
 * @version dato
 */
```

NB!

JavaDoc krever at klassen deklareres som **public class**.

Kommentarer for metoder ser slik ut:

```
/**
 * Én setning som kort beskriver metoden
 * Ytterligere kommentarer
 *      ⋮
 * @param  navn1 Kort beskrivelse av parameteren
 * @param  navn2 Kort beskrivelse av parameteren
 * @return Kort beskrivelse av returverdien
 * @see    navn3
 */
```

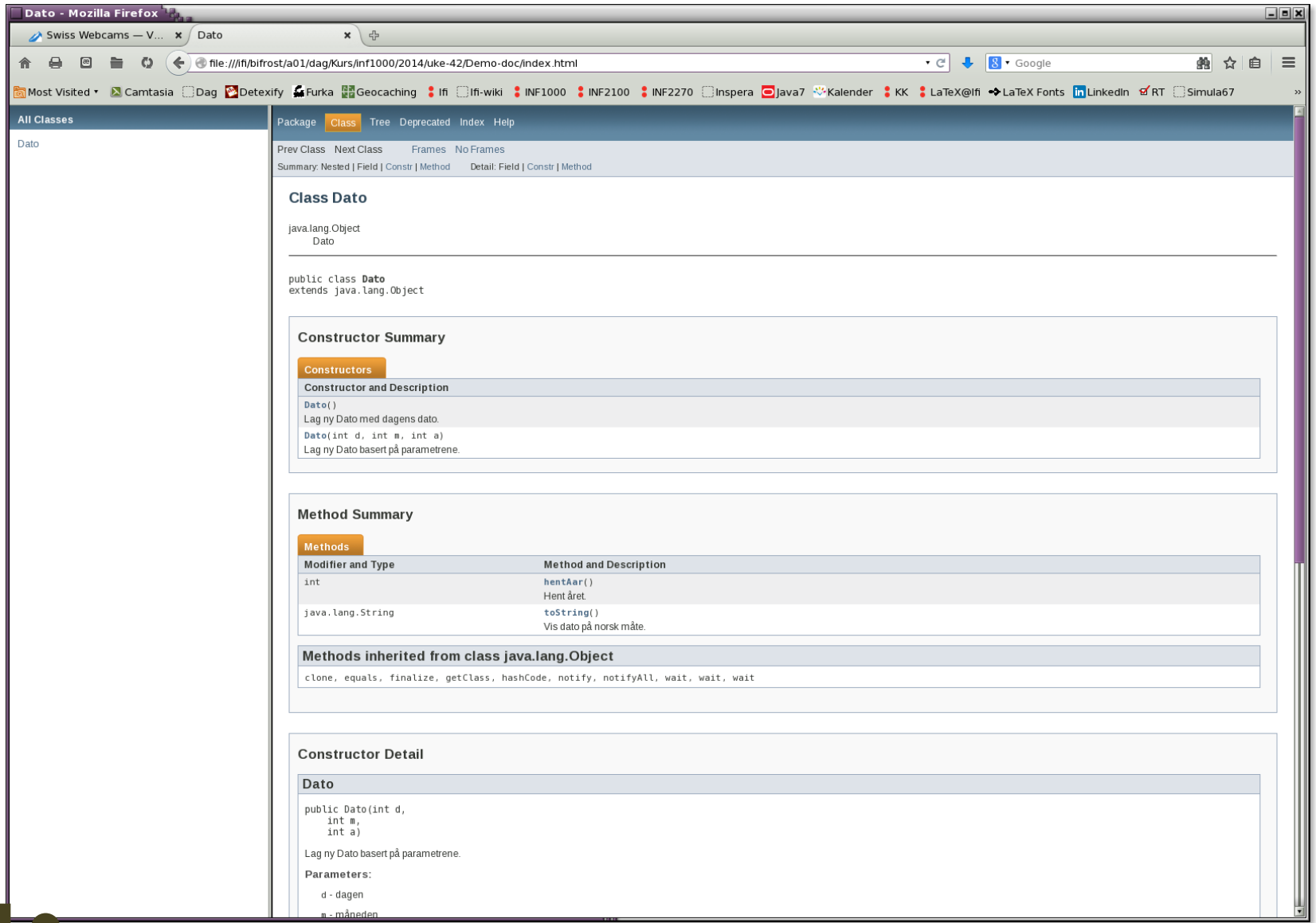
NB!

JavaDoc krever at metoden deklareres som **public**.

Demo/Dato.java

```
/**
 * En dato

 * @author: dag@ifi.uio.no
 * @version: 2014-10-14
 */

import java.util.Calendar;

public class Dato {
    private int dag, mnd, aar;

    /**
     * Lag ny Dato basert p&aring; parametrene.
     * @param d dagen
     * @param m m&aring;neden
     * @param a &aring;ret
     */
    public Dato(int d, int m, int a) {
        dag = d;  mnd = m;  aar = a;
    }

    /**
     * Lag ny Dato med dagens dato.
     */
    public Dato() {
        Calendar idag = Calendar.getInstance();
        dag = idag.get(Calendar.DAY_OF_MONTH);
        mnd = idag.get(Calendar.MONTH)+1;  // NB! Husk +1!
        aar = idag.get(Calendar.YEAR);
    }
```

Resultatet


Dato - Mozilla Firefox
Swiss Webcams — V...
Dato
file:///ifi/bifrost/a01/dag/Kurs/inf1000/2014/uke-42/Demo-doc/index.html
All Classes
Dato
Package Class Tree Deprecated Index Help
Prev Class Next Class Frames No Frames
Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method
Class Dato
java.lang.Object
Dato
public class Dato
extends java.lang.Object
Constructor Summary
Constructors
Constructor and Description
Dato()
Lag ny Dato med dagens dato.
Dato(int d, int m, int a)
Lag ny Dato basert på parametrene.
Method Summary
Methods
Modifier and Type
Method and Description
int
hentAar()
Hent året.
java.lang.String
toString()
Vis dato på norsk måte.
Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait
Constructor Detail
Dato
public Dato(int d,
int m,
int a)
Lag ny Dato basert på parametrene.
Parameters:
d - dagen
m - måneden

Kommentarer i koden

Kommentarer i koden kan brukes for å forklare detaljer eller grunner til at vi gjør noe uvanlig. Det bør ikke være for mange slike.

- maaned = idag.get(MONTH)+1; // Husk januar==0!
- /* Denne koden er kommentert vekk.
  Den ble brukt til å finne en feil i testdata
  men trengs ikke lenger.
  ⋮
  */

Selvreferansen this

Noen ganger trenger vi å referere til det objektet vi selv er, for eksempel

```
class C {
  private C p = this;
      :
}
```

Nå vil p peke på seg selv.